ATION PAGE

Form Approved
OPM No. 0704-0188

AD-A223 580

[reporting burden text, illegible] ... Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 31 Jan 90 to 31 Jan 91	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: TeleSoft and BBN, TC2000 pSOS Ada 4.0 for BBN TC2000/nX 1.0 (Host) to TC2000/pSOS (Target), 900131I1.10268

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, FEDERAL REPUBLIC OF GERMANY

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AV Industrieanlagen-Betriebsgeselschaft
Dept. SZT
Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER

AVF-IABG-067

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

TeleSoft and BBN, TC200 pSOS Ada 4.0, Ottobrunn West Germany, BBN TC2000 under nX 1.0 (Host) to BBN TC2000 under TC2000 pSOS 0.0.2 (Target), ACVC 1.10.

DTIC
ELECTE
JUN 27 1990
S E D

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-01

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #900131I1.10268
TeleSoft and BBN
TC2000 pSOS Ada 4.0 for
BBN TC2000/nX 1.0 to TC2000/pSOS

Completion of On-Site Testing:
31 January 1990

Prepared By:
IABG mbH, Abt. SZT
Einsteinstrasse 20
D-8012 Ottobrunn
West Germany

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

90 06 25 118

Ada   Compiler Validation Summary Report:

Compiler Name: TC2000 pSOS Ada Version 4.0

Certificate Number: #900131I1.10268

Host: BBN TC2000 under nX 1.0

Target: BBN TC2000 under TC2000 pSOS 0.8.2

Testing Completed 31 January 1990 Using ACVC. 1.10

This report has been reviewed and is approved.

_____

IABG mbH, Abt. SZT
Dr. S. Heilbrunner
Einsteinstr. 20
D-8012 Ottobrunn
West Germany

_____
Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria, VA   22311

_____
Ada Joint Program Office
Dr John Solomond
Director
Department of Defense
Washington DC   20301

TABLE OF CONTENTS

1

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

2

. To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

. To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

. To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO).

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure c: this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

> Ada Information Clearinghouse
> Ada Joint Program Office
> OUSDRE
> The Pentagon, Rm 3D-139 (Fern Street)
> Washington DC 20301-3081

or from:
> IABG mbH, Abt. SZT
> Einsteinstr. 20
> D-8012 Ottobrunn
> West Germany

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

> Ada Validation Organization
> Institute for Defense Analyses
> 1801 North Beauregard Street
> Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language,
    ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint
    Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech,
    Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC   The Ada Compiler Validation Capability. The set of Ada
                programs that tests the conformity of an Ada compiler to the
                Ada programming language.

Ada   An Ada Commentary contains all information relevant to the
Commentary   point addressed by a comment on the Ada Standard. These
                comments are given a unique identification number having the
                form AI-ddddd.

Ada Standard   ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant   The agency requesting validation.

AVF   The Ada Validation Facility. The AVF is responsible for
                conducting compiler validations according to procedures
                contained in the Ada Compiler Validation Procedures and
                Guidelines.

AVO   The Ada Validation Organization. The AVO has oversight
                authority over all AVF practices for the purpose of
                maintaining a uniform process for validation of Ada
                compilers. The AVO provides administrative and technical
                support for Ada validations to ensure consistent practices.

Compiler   A processor for the Ada language. In the context of this
                report, a compiler is any language processor, including
                cross-compilers, translators, and interpreters.

Failed test   An ACVC test for which the compiler generates a result that
                demonstrates nonconformity to the Ada Standard.

Host   The computer on which the compiler resides.

Inapplicable An ACVC test that uses features of the language that a
test   compiler is not required to support or may legitimately
              support in a way other than the one expected by the test.

Passed test   An ACVC test for which a compiler generates the expected
              result.

Target   The computer which executes the code generated by the
              compiler.

Test   A program that checks a compiler's conformity regarding a
              particular feature or a combination of features to the Ada
              Standard. In the context of this report, the term is used to
              designate a single test, which may comprise one or more
              files.

Withdrawn   An ACVC test found to be incorrect and not used to check
test   conformity to the Ada Standard. A test may be incorrect
              because it has an invalid test objective, fails to meet its
              test objective, or contains illegal or erroneous use of the
              language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC
contains both legal and illegal Ada programs structured into six test
classes: A, B, C, D, E, and L. The first letter of a test name identifies
the class to which it belongs. Class A, C, D, and E tests are executable,
and special program units are used to report their results during
execution. Class B tests are expected to produce compilation errors.
Class L tests are expected to produce errors because of the way in which a
program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada
programs with certain language constructs which cannot be verified at run
time. There are no explicit program components in a Class A test to check
semantics. For example, a Class A test checks that reserved words of
another language (other than those already reserved in the Ada language)
are not treated as reserved words by an Ada compiler. A Class A test is
passed if no errors are detected at compile time and the program executes
to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage.
Class B tests are not executable. Each test in this class is compiled and
the resulting compilation listing is examined to verify that every syntax
or semantic error in the test is detected. A Class B test is passed if
every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and tests. However, some tests contain values that require the test to be

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: TC2000 pSOS Ada Version 4.0

ACVC Version: 1.10

Certificate Number: #900131I1.10268

Host Computer:

Machine:	BBN TC2000
Operating System:	nX 1.0
Memory Size:	16 MegaBytes

Target Computer:

Machine:	BBN TC2000
Operating System:	TC2000 pSOS 0.8.2
Memory Size:	4 MegaBytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of
a compiler in those areas of the Ada Standard that permit implementations
to differ. Class D and E tests specifically check for such implementation
differences.   However, tests in other classes also characterize an
implementation. The tests demonstrate the following characteristics:

a. Capacities.

    1) The   compiler   correctly   processes   a   compilation
        containing 723 variables in the same declarative part. (See
        test D29002K.)

    2) The   compiler   correctly   processes   tests   containing
        loop statements nested to 65 levels. (See tests D55A03A..H
        (8 tests).)

    3) The   compiler   correctly   processes   tests   containing
        block statements nested to 65 levels. (See test D56001B.)

    4) The   compiler   correctly   processes   tests   containing
        recursive procedures separately compiled as subunits nested to
        17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

    1) This implementation supports the additional predefined types
        SHORT_INTEGER, SHORT_SHORT_INTEGER and LONG_FLOAT in the
        package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

    The order in which expressions are evaluated and the time at which
    constraints are checked are not defined by the language. While
    the ACVC tests do not specifically attempt to determine the order
    of evaluation of expressions, test results indicate the following:

    1) Some   of   the   default   initialization   expressions
        for record components are evaluated before any value is
        checked for membership in a component's subtype. (See test
        C32117A.)

    2) Assignments for subtypes are performed with the same
        precision as the base type. (See test C35712B.)

    3) This implementation uses no extra bits for extra precision
        and uses no extra bits for extra range. (See test C35903A.)

4) NUMERIC_ERROR is raised for predefined and largest integer comparison tests, and NUMERIC_ERROR is raised for predefined and largest integer membership tests, and no exception is raised for smallest integer membership tests when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

1) The method used for rounding to integer is round to even. (See tests C46012A..Z (26 tests).)

2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z (26 tests).)

3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception under the specific circumstances test C36003A.

2) NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

3) NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

10

4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

6) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

8) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR. (See test E52103Y.)

f. Discriminated types.

1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

1) In the evaluation of a multi-dimensional aggregate, the test results indicate that index subtype checks are made as choices are evaluated. (See tests C43207A and C43207B.)

2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

1) The pragma INLINE is not supported for procedures or functions. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

This implementation creates a dependence between a generic body and those units which instatiate it. As allowed by AI-408/11, if the body is compiled after a unit that instantiates it, then that unit becomes obsolete.

1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

1) The package SEQUENTIAL_IO can be instantiated with uncon-strained array types or record types with discriminants with-out defaults. (See tests AE2101C, EE2201D, and EE2201E.)

2) The package DIRECT_IO cannot be instantiated with uncon-strained array types or record types with discriminants with-out defaults. (See tests AE2101H, EE2401D, and EE2401G.)

3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

8) RESET and DELETE operations are supported for· text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

9) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

10) Temporary sequential files are given names and not deleted when closed. (See test CE2108A.)

11) Temporary direct files are not given names and not deleted when closed. (See test CE2108C.)

12) Temporary text files are not given names and not deleted when closed. (See test CE3112A.)

13) More than one internal file can be associated with each external file for sequential files when reading only. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

13

14) More than one internal file can be associated with each external file for direct files when reading only. (See tests CE2107F..H (3 tests), CE2110D and CE2111H.)

15) More than one internal file can be associated with each external file for text files when reading only (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 312 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 12 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	128	1130	2018	17	24	44	3361
Inapplicable	1	8	297	0	4	2	312
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	2	3	4	5	6	7	8	9	10	11	12	13	14	TOTAL
Passed	198	573	545	244	172	99	161	332	129	36	250	341	281	3361
N/A	14	76	135	4	0	0	5	0	8	0	2	28	40	312
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

(header: TEST CHAPTER)

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10
at the time of this validation:

```
E28005C    A39005G    B97102E    C97116A    BC3009B    CD2A62D
CD2A63A    CD2A63B    CD2A63C    CD2A63D    CD2A66A    CD2A66B
CD2A66C    CD2A66D    CD2A73A    CD2A73B    CD2A73C    CD2A73D
CD2A76A    CD2A76B    CD2A76C    CD2A76D    CD2A81G    CD2A83G
CD2A84N    CD2A84M    CD5011O    CD2B15C    CD7205C    CD2D11B
CD5007B    ED7004B    ED7005C    ED7005D    ED7006C    ED7006D
CD7105A    CD7203B    CD7204B    CD7205D    CE2107I    CE3111C
CE3301A    CE3411B
```

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features
that a compiler is not required by the Ada Standard to support. Others may
depend on the result of another test that is either inapplicable or
withdrawn. The applicability of a test to an implementation is considered
each time a validation is attempted. A test that is inapplicable for one
validation attempt is not necessarily inapplicable for a subsequent
attempt. For this validation attempt, 312 tests were inapplicable for

   a. The following 201 tests are not applicable because they have
       floating-point type declarations requiring more digits than
       SYSTEM.MAX_DIGITS:

```
       C24113L..Y (14 tests)      C35705L..Y (14 tests)
       C35706L..Y (14 tests)      C35707L..Y (14 tests)
       C35708L..Y (14 tests)      C35802L..Z (15 tests)
       C45241L..Y (14 tests)      C45321L..Y (14 tests)
```

C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

b. C35508I, C35508J, C35508M, and C35508N are not applicable because they include enumeration representation clauses for BOOLEAN types in which the representation values are other than (FALSE => 0, TRUE => 1). Under the terms of AI-00325, this implementation is not required to support such representation clauses.

c. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

d. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because they acquire a value of SYSTEM.MAX_MANTISSA greater than 32.

e. C52008B is not applicable because this implementation generates code to calculate the maximum object size for type REC2 at run time which yields a number exceeding INTEGER'LAST and raises NUMERIC_ERROR.

f. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

g. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

h. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

i. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

j. CA2009C, CA2009F, BC3204C and BC3205D are not applicable because this implementation creates a dependence between a generic body and those units which instantiate it (See Section 2.2.i and Appendix F of the Ada Standard).

k. LA3004A, EA3004C, and CA3004E are not applicable because this implementation does not support pragma INLINE for procedures.

17

l. LA3004B, EA3004D, and CA3004F are not applicable because this implementation does not support pragma INLINE for functions.

m. CD1009C, CD2A41A..B (2 tests), CD2A41E and CD2A42A..J (10 tests) are not applicable because of restrictions on 'SIZE length clauses for floating point types.

n. CD1C04E is not applicable because this implementation does not support component clauses specifying more than 8 bits for boolean components of a record.

o. CD2A61I..J (2 tests) are not applicable because of restrictions on 'SIZE length clauses for array types.

p. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because of restrictions on 'SIZE length clauses for access types.

q. AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

r. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

s. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

t. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

u. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

v. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

w. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

x. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

y. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

z. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

aa. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

18

ab. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

ac. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

ad. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

ae. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

af. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

ag. CE2107B..E (4 tests), CE2107L, CE2110B, and CE2111D are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

ah. CE2107G..H (2 tests), CE2110D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

ai. CE3102E is inapplicable because text file CREATE with IN_FILE mode is supported by this implementation.

aj. CE3102F is inapplicable because text file RESET is supported by this implementation.

ak. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.

al. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.

am. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

an. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is supported by this implementation.

ao. CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

19

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 12 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B71001E	B71001K	B71001Q	B71001W	BA2001C
BA2001E	BA3006A	BA3006B	BA3007B	BA3008A
BA3008B	BA3013A (6 and 7M)			

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, no test results for ACVC Version 1.10 produced with the TC2000 pSOS Ada System was submitted to the AVF by the applicant for review because the applicant certified that they would be identical to the results of the TC2000 Ada System (Certificate Number #89112311.10218).

3.7.2 Test Method

Testing of the TC2000 pSOS Ada Compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host Computer:	BBN TC2000
Host Operating System:	nX 1.0
Target Computer:	BBN TC2000
Target Operating System:	TC2000 pSOS 0.8.2

Host and target computers were part of a multiprocessor machine. Different operating systems were used for the processors selected for this validation. Communication between the host and target processors was achieved by shared memory.

A cartridge containing the customized test suite was loaded onto a SUN-3 and transferred via Ethernet to the host computer.

The compiler was tested using command scripts provided by TeleSoft and reviewed by the validation team. The tests were compiled using the command

    ada   -V 2000 -S -W +enable -W +driver <test file>

and linked with the command

    ald   -V 2000 -S -W +link_target=bbn <main unit>

The -L qualifier was added to the compiler call for class B and E tests. See Appendix E for explanation of compiler and linker switches.

Tests were compiled, linked, and executed (as appropriate) using two computers. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at TeleSoft, San Diego, USA, and was completed on 31 January 1990.

21

APPENDIX A

DECLARATION OF CONFORMANCE

TeleSoft and BBN have submitted the following Declaration of Conformance concerning the TC2000 Ada Compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: TELESOFT
Ada Validation Facility: IABG, Dept. SZT, D-8012 Ottobrunn
Ada Compiler Validation Capability (ACVC) Version: 1.10
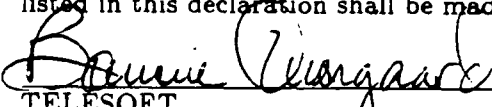
Base Configuration

Base Compiler Name: TC2000 pSOS$^{+™}$ Ada
Version: 4.0

Host Architecture ISA: BBN TC2000
OS & VER #:   nX 1.0

Target Architecture ISA: BBN TC2000
OS & VER #:   TC2000 pSOS$^{+™}$ 0.8.2

Implementor's Declaration
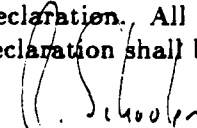
I, the undersigned, representing TELESOFT, declare that TELESOFT has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration. I declare that BBN ADVANCED COMPUTERS, INC., is TELESOFT's licensee of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the licensee's corporate name.

_____   Date: 2-1-90
TELESOFT
Raymond A. Parra, Vice President and General Counsel

Licensee's Declaration

I, the undersigned, representing BBN ADVANCED COMPUTERS, INC., declare that BBN ADVANCED COMPUTERS, INC., has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the licensee's corporate name.

_____   Date: 1 Feb. 1990
BBN ADVANCED COMPUTERS, INC.
Name and Title: Richard Schooler , Senior Engineer

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the TC2000 pSOS Ada Compiler, as described in this Appendix, are provided by TeleSoft. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

    ...

        type INTEGER is range -2147483648 .. 2147483647;
        type SHORT_INTEGER is range -32768 .. 32767;
        type SHORT_SHORT_INTEGER is range -128..127;

        type FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;
        type LONG_FLOAT is digits 15
            range -8.98846567431158E+307 .. 8.98846567431158E+307;

        type DURATION is delta 2#1.0#E-14 range -86400.0 .. 86400.0;

    ...

end STANDARD;
```

ATTACHMENT B

Appendix F OF THE Ada LANGUAGE REFERENCE MANUAL

1. Implementation Dependent Pragmas
2. Implementation Dependent Attributes
3. Specification of Package SYSTEM
4. Restrictions on representation clauses
5. Implementation dependent naming
6. Interpretation of expressions in address clauses
7. Restrictions on unchecked conversions
8. I/O Package characteristics

Appendix F

1. Implementation Dependent Pragmas

pragma COMMENT(<string_literal>);
It may only appear within a compilation unit.
The pragma comment has the effect of embedding the given
sequence of characters in the object code of the compilation unit.

pragma LINKNAME(<subprogram_name>, <string_literal>);
It may appear in any declaration section of a unit.
This pragma must also appear directly after an interface pragma
for the same <subprogram_name>. The pragma linkname has the
effect of making string_literal apparent to the linker.

pragma INTERRUPT(Function_Mapping);
It may only appear immediately before a simple accept statement,
a while loop directly enclosing only a single accept statement,
or a select statement that includes an interrupt accept alternative.
The pragma interrupt has the effect that entry calls to the
associated entry, on behalf of an interrupt, are made with a
reduced call overhead.

pragma IMAGES(<enumeration_type>,Deferred) or
pragma IMAGES(<enumeration_type>,Immediate);
It may only appear within a compilation unit.
The pragma images controls the creation and allocation of
the image table for a specified enumeration type. The
default is Deferred, which saves space in the literal pool
by not creating an image table for an enumeration type
unless the 'Image, 'Value, or 'Width attribute for the type
is used. If one of these attributes is used, an image table
is generated in the literal pool of the compilation unit in
which the attribute appears. If the attributes are used in
more than one compilation unit. more than one image table is
generated. eliminating the benefits of deferring the table.

pragma SUPPRESS_ALL;
It may appear anywhere that a Suppress pragma may appear as
defined by the Language Reference Manual. The pragma
Suppress_All has the effect of turning off all checks
defined in section 11.7 of the Language Reference Manual.
The scope of applicablility of this pragma is the same as
that of the pre-defined pragma Suppress.

2. Implementation Dependent Attributes

'Offset Attribute

'Offset along with the attribute 'Address, facilitates machine code
insertions. For a prefix P that denotes a declared parameter
object, P'Offset yields the statically known portion of the
address of the first of the storage units allocated to P. The
value is the object's offset relative to a base register and is
of type Long_Integer.

INTEGER ATTRIBUTES

'Extended_Image Attribute

Usage: X'Extended_Image(Item,Width,Base,Based,Space_IF_Positive)

Returns the image associated with Item as per the Text_Io definition.
The Text_Io definition states that the value of Item is an integer
literal with no underlines, no exponent, no leading zeros
(but a single zero for the zero value) and a minus sign if negative.
If the resulting sequence of characters to be output has fewer than
Width characters then leading spaces are first output to make up
the difference. (LRM 14.3.7:10,14.3.7:11)

For a prefix X that is a discrete type or subtype; this attribute
is a function that may have more than one parameter. The parameter
Item must be an integer value. The resulting string is
without underlines, leading zeros, or trailing spaces.

Parameter Descriptions:

Item   — The user specifies the item that he wants the
              image of and passes it into the function. This
              parameter is required.

Width   — The user may specify the minimum number.of
              characters to be in the string that is returned.
              If no width is specified then the default (0) is
              assumed.

Base   — The user may specify the base that the image is
              to be displayed in. If no base is specified then
              the default (10) is assumed.

Based   — The user may specify whether he wants the string
              returned to be in base notation or not. If no
              preference is specified then the default (false)
              is assumed.

Space_If_Positive — The user may specify whether or not the sign bit
              of a positive integer is included in the string
              returned. If no preference is specified then
              the default (false) is assumed.

Examples:

Suppose the following subtype was declared:

Subtype X is Integer Range -10..16;

Then the following would be true:

```
X'Extended_Image(5)                   = "5"
X'Extended_Image(5,0)                 = "5"
X'Extended_Image(5,2)                 = " 5"
X'Extended_Image(5,0,2)               = "101"
X'Extended_Image(5,4,2)               = " 101"
X'Extended_Image(5,0,2,True)          = "2#101#"
X'Extended_Image(5,0,10,False)        = "5"
X'Extended_Image(5,0,10,False,True)   = " 5"
X'Extended_Image(-1,0,10,False,False) = "-1"
X'Extended_Image(-1,0,10,False,True)  = "-1"
X'Extended_Image(-1,1,10,False,True)  = "-1"
X'Extended_Image(-1,0,2,True,True)    = "-2#1#"
X'Extended_Image(-1,10,2,True,True)   = "    -2#1#"
```

Appendix F, Cont.

'Extended_Value Attribute

Usage: X'Extended_Value(Item)

Returns the value associated with Item as per the Text_Io definition.
The Text_Io definition states that given a string, it reads an
integer value from the beginning of the string. The value returned
corresponds to the sequence input. (LRM 14.3.7:14)

For a prefix X that is a discrete type or subtype; this attribute
is a function with a single parameter. The actual parameter Item
must be of predefined type string. Any leading or trailing
spaces in the string X are ignored. In the case where an illegal
string is passed, a CONSTRAINT_ERROR is raised.

Parameter Descriptions:

Item -- The user passes to the function a parameter of the
        predefined type string. The type of the returned
        value is the base type X.

Examples:

Suppose the following subtype was declared:

  Subtype X is Integer Range -10..16;

Then the following would be true:

    X'Extended_Value("5")   = 5
    X'Extended_Value(" 5")   = 5
    X'Extended_Value("2#101#")   = 5
    X'Extended_Value("-1")   = -1
    X'Extended_Value(" -1")   = -1

'Extended_Width Attribute

Usage: X'Extended_Width(Base,Based,Space_If_Positive)

Returns the width for subtype of X.

For a prefix X that is a discrete subtype; this attribute
is a function that may have multiple parameters. This attribute
yields the maximum image length over all values of the type
or subtype X.

Parameter Descriptions:

Base — The user specifies the base for which the width
will be calculated. If no base is specified
then the default (10) is assumed.

Based — The user specifies whether the subtype is stated
in based notation. If no value for based is
specified then the default (false) is assumed.

Space_If_Positive — The user may specify whether or not the sign bit
of a positive integer is included in the string
returned. If no preference is specified then the
default (false) is assumed.

Examples:

Suppose the following subtype was declared:

Subtype X is Integer Range -10..16;

Then the following would be true:

X'Extended_Width	= 3 —	"-10"
X'Extended_Width(10)	= 3 —	"-10"
X'Extended_Width(2)	= 5 —	"10000"
X'Extended_Width(10,True)	= 7 —	"-10#10#"
X'Extended_Width(2,True)	= 8 —	"2#10000#"
X'Extended_Width(10,False,True)	= 3 —	" 16"
X'Extended_Width(10,True,False)	= 7 —	"-10#10#"
X'Extended_Width(10,True,True)	= 7 —	" 10#16#"
X'Extended_Width(2,True,True)	= 9 —	" 2#10000#"
X'Extended_Width(2.False,True)	= 6 —	" 10000"

ENUMERATION ATTRIBUTES

'Extended_Image Attribute

Usage: X'Extended_Image(Item,Width,Uppercase)

Returns the image associated with Item as per the Text_Io definition.
The Text_Io definition states that given an enumeration literal,
it will output the value of the enumeration literal (either an
identifier or a character literal). The character case parameter
is ignored for character literals. (LRM 14.3.9:9)

For a prefix X that is a discrete type or subtype: this attribute
is a function that may have more that one parameter. The parameter
Item must be an enumeration value. The image of an enumeration
value is the corresponding identifier which may have character case
and return string width specified.

Parameter Descriptions:

Item — The user specifies the item that he wants the image of and
passes it into the function. This parameter is required.

Width — The user may specify the minimum number of characters to
be in the string that is returned. If no width is
specified then the default (0) is assumed. If the Width
specified is larger than the image of Item. then the
return string is padded with trailing spaces: if the
Width specified is smaller than the image of Item then
the default is assumed and the image of the enumeration
value is output completely.

Uppercase — The user may specify whether the returned string is in
uppercase characters. In the case of an enumeration
type where the enumeration literals are character
literals. the Uppercase is ignored and the case
specified by the type definition is taken. If no
preference is specified then the default (true) is
assumed.

Appendix F, Cont.

Examples:

Suppose the following types were declared:

Type X is (red, green, blue, purple);
Type Y is ('a', 'B', 'c', 'D');

Then the following would be true:

X'Extended_Image(red)	= "RED"
X'Extended_Image(red, 4)	= "RED "
X'Extended_Image(red,2)	= "RED"
X'Extended_Image(red,0,false)	= "red"
X'Extended_Image(red,10,false)	= "red "
Y'Extended_Image('a')	= "'a'"
Y'Extended_Image('B')	= "'B'"
Y'Extended_Image('a',6)	= "'a' "
Y'Extended_Image('a',0,true)	= "'a'"

'Extended_Value Attribute

Usage: X'Extended_Value(Item)

Returns the image associated with Item as per the Text_Io definition.
The Text_Io definition states that it reads an enumeration value
from the beginning of the given string and returns the value of
the enumeration literal that corresponds to the sequence input.
(LRM 14.3.9:11)

For a prefix X that is a discrete type or subtype; this attribute
is a function with a single parameter. The actual parameter Item
must be of predefined type string. Any leading or trailing
spaces in the string X are ignored. In the case where an illegal
string is passed, a CONSTRAINT_ERROR is raised.

Parameter Descriptions:

Item -- The user passes to the function a parameter of the
        predefined type string. The type of the returned
        value is the base type of X.

Appendix F, Cont.

Examples:

Suppose the following type was declared:

Type X is (red, green, blue, purple);

Then the following would be true:

X'Extended_Value("red")	= red
X'Extended_Value(" green")	= green
X'Extended_Value(" Purple")	= purple
X'Extended_Value(" GreEn ")	= green

'Extended_Width Attribute

Usage: X'Extended_Width

Returns the width for subtype of X.

For a prefix X that is a discrete type or subtype; this attribute
is a function. This attribute yields the maximum image length over
all values of the enumeration type or subtype X.

Parameter Descriptions:

There are no parameters to this function. This function
returns the width of the largest (width) enumeration literal
in the enumeration type specified by X.

Examples:

Suppose the following types were declared:

Type X is (red, green, blue, purple);
Type Z is (X1, X12, X123, X1234);

Then the following would be true:

X'Extended_Width	= 6 — "purple"
Z'Extended_Width	= 5 — "X1234"

FLOATING POINT ATTRIBUTES

'Extended_Image Attribute

Usage: X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)

Returns the image associated with Item as per the Text_Io definition.
The Text_Io definition states that it outputs the value of the
parameter Item as a decimal literal with the format defined by the
other parameters. If the value is negative then a minus sign
is included in the integer part of the value of Item. If Exp
is 0 then the integer part of the output has as many digits as
are needed to represent the integer part of the value of Item or
is zero if the value of Item has no integer part. (LRM 14.3.8:13,
14.3.8:15)

For a prefix X that is a discrete type or subtype; this attribute
is a function that may have more than one parameter. The parameter
Item must be a Real value. The resulting string is
without underlines or trailing spaces.

Parameter Descriptions:

Item — The user specifies the item that he wants the image of and
passes it into the function. This parameter is required.

Fore — The user may specify the minimum number of characters for
the integer part of the decimal representation in the
return string. This includes a minus sign if the
value is negative and the base with the '#' if based
notation is specified. If the integer part to be output
has fewer characters than specified by Fore, then leading
spaces are output first to make up the difference. If no
Fore is specified then the default (2) value is assumed.

Aft — The user may specify the minimum number of decimal digits
after the decimal point to accommodate the precision desired.
If the delta of the type or subtype is greater than
0.1 then Aft is one. If no Aft is specified then the
default (X'Digits-1) is assumed. If based notation is
specified the trailing '#' is included in aft.

Exp — The user may specify the minimum number of digits in the
exponent: the exponent consists of a sign and the exponent,
possibly with leading zeros. If no Exp is specified then
the default (3) is assumed. If Exp is 0 then no exponent
is used.

Base — The user may specify the base that the image is to be
displayed in. If no base is specified then the default
(10) is assumed.

Based — The user may specify whether he wants the string returned
to be in based notation or not. If no preference is
specified then the default (false) is assumed.

Examples:

Suppose the following type was declared:

Type X is digits 5 range -10.0 .. 16.0;

Then the following would be true:

```
X'Extended_Image(5.0)              = " 5.0000E+00"
X'Extended_Image(5.0,1)            = "5.0000E+00"
X'Extended_Image(-5.0,1)           = "-5.0000E+00"
X'Extended_Image(5.0,2,0)          = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)        = " 5.0"
X'Extended_Image(5.0,2,0,0,2)      = "101.0"
X'Extended_Image(5.0,2,0,0,2,True) = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True) = "2#1.1#E+02"
```

'Extended_Value Attribute

Usage: X'Extended_Value(Item)

Returns the value associated with Item as per the Text_Io definition.
The Text_Io definition states that it skips any leading zeros,
then reads a plus or minus sign if present then reads the string
according to the syntax of a real literal. The return value is
that which corresponds to the sequence input. (LRM 14.3.8:9,
14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute
is a function with a single parameter. The actual parameter Item
must be of predefined type string. Any leading or trailing
spaces in the string X are ignored. In the case where an illegal
string is passed. a CONSTRAINT_ERROR is raised.

Appendix F, Cont.

Parameter Descriptions:

Item — The user passes to the function a parameter of the
predefined type string. The type of the returned
value is the base type of the input string.

Examples:

Suppose the following type was declared:

Type X is digits 5 range -10.0 .. 16.0;

Then the following would be true:

X'Extended_Value("5.0")   = 5.0
X'Extended_Value("0.5E1")   = 5.0
X'Extended_Value("2#1.01#E2")   = 5.0

'Extended_Digits Attribute

Usage: X'Extended_Digits(Base)

Returns the number of digits using base in the mantissa of model
numbers of the subtype X.

Parameter Descriptions:

Base — The user may specify the base that the subtype is
defined in. If no base is specified then the default
(10) is assumed.

Examples:

Suppose the following type was declared:

Type X is digits 5 range -10.0 .. 16.0;

Then the following would be true:

X'Extended_Digits   = 5

FIXED POINT ATTRIBUTES

'Extended_Image Attribute

Usage: X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)

Returns the image associated with Item as per the Text_Io definition.
The Text_Io definition states that it outputs the value of the
parameter Item as a decimal literal with the format defined by the
other parameters. If the value is negative then a minus sign
is included in the integer part of the value of Item. If Exp
is 0 then the integer part of the output has as many digits as
are needed to represent the integer part of the value of Item or
is zero if the value of Item has no integer part. (LRM 14.3.8:13,
14.3.8:15)

For a prefix X that is a discrete type or subtype: this attribute
is a function that may have more than one parameter. The parameter
Item must be a Real value. The resulting string is
without underlines or trailing spaces.

Parameter Descriptions:

Item — The user specifies the item that he wants the image of and
         passes it into the function. This parameter is required.
Fore — The user may specify the minimum number of characters for
         the integer part of the decimal representation in the
         return string. This includes a minus sign if the
         value is negative and the base with the '#' if based
         notation is specified. If the integer part to be output
         has fewer characters than specified by Fore, then leading
         spaces are output first to make up the difference. If no
         Fore is specified then the default (2) value is assumed.
Aft   — The user may specify the minimum number of decimal digits
         after the decimal point to accommodate the precision desired.
         · If the delta of the type or subtype is greater than
         0.1 then Aft is one. If no Aft is specified then the
         default (X'Digits-1) is assumed. If based notation is
         specified the trailing '#' is included in aft.
Exp   — The user may specify the minimum number of digits in the
         exponent: the exponent consists of a sign and the exponent,
         possibly with leading zeros. If no Exp is specified then
         the default (3) is assumed. If Exp is 0 then no exponent
         is used.

Appendix F, Cont.

Base — The user may specify the base that the image is to be displayed in. If no base is specified then the default (10) is assumed.

Based — The user may specify whether he wants the string returned to be in based notation or not. If no preference is specified then the default (false) is assumed.

Examples:

Suppose the following type was declared:

Type X is delta 0.1 range -10.0 .. 17.0;

Then the following would be true:

X'Extended_Image(5.0)	= " 5.00E+00"
X'Extended_Image(5.0,1)	= "5.00E+00"
X'Extended_Image(-5.0,1)	= "-5.00E+00"
X'Extended_Image(5.0,2.0)	= " 5.0E+00"
X'Extended_Image(5.0,2,0,0)	= " 5.0"
X'Extended_Image(5.0,2,0,0,2)	= "101.0"
X'Extended_Image(5.0,2,0,0,2,True)	= "2#101.0#"
X'Extended_Image(5.0,2,2.3,2,True)	= "2#1.1#E+02"

'Extended_Value Attribute

Usage: X'Extended_Value(Image)

Returns the value associated with Item as per the Text_Io definition. The Text_Io definition states that it skips any leading zeros. then reads a plus or minus sign if present then read the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed. a CONSTRAINT_ERROR is raised.

Appendix F, Cont.

Parameter Descriptions:

Image -- The user passes to the function a parameter of the
predefined type string. The type of the returned
value is the base type of the input string.

Examples:

Suppose the following type was declared:

Type X is delta 0.1 range -10.0 .. 17.0;

Then the following would be true:

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Fore Attribute

Usage: X'Extended_Fore(Base,Based)

Returns the minimum number of characters required for the integer
part of the based representation of X.

Parameter Descriptions:

Base -- The user may specify the base that the subtype would be
displayed in. If no base is specified then the default
(10) is assumed.
Based -- The user may specify whether he wants the string returned
·to be in based notation or not. If no preference is
specified then the default (false) is assumed.

Appendix F, Cont.

Examples:

Suppose the following type was declared:

Type X is delta 0.1 range -10.0 .. 17.1;

Then the following would be true:

X'Extended_Fore   = 3 — "-10"
X'Extended_Fore(2)   = 6 — " 10001"

'Extended_Aft Attribute

Usage: X'Extended_Aft(Base,Based)

Returns the minimum number of characters required for the fractional part of the based representation of X.

Parameter Descriptions:

Base — The user may specify the base that the subtype would be
         displayed in. If no base is specified then the default
         (10) is assumed.
Based — The user may specify whether he wants the string returned
         to be in based notation or not. If no preference is
         specified then the default (false) is assumed.

Examples:

Suppose the following type was declared:

Type X is delta 0.1 range -10.0 .. 17.1;

Then the following would be true:

X'Extended_Aft   = 1 — "1" from 0.1
X'Extended_Aft(2)   = 4 — "0001" from 2#0.0001#

Appendix F, Cont.

3. Specification of Package SYSTEM

with Unchecked_Conversion;

package System is

```
--================================
-- CUSTOMIZABLE VALUES
--================================


type Name    is (TeleGen2);

System_Name  : constant name := TeleGen2;

Memory_Size  : constant := (2 ** 31) -1; --Available memory, in storage units
Tick         : constant := 1.0 / 100.0; --Basic clock rate, in seconds


--================================================================
-- NON-CUSTOMIZABLE, IMPLEMENTATION-DEPENDENT VALUES
--================================================================


Storage_Unit : constant := 8;
Min_Int      : constant := -(2 ** 31);
Max_Int      : constant := (2 ** 31) - 1;
Max_Digits   : constant := 15;
Max_Mantissa : constant := 31;
Fine_Delta   : constant := 1.0 / (2 ** Max_Mantissa);

subtype Priority is Integer Range 0 .. 63;


--================================
-- ADDRESS TYPE SUPPORT
--================================


type Memory is private;
type Address is access Memory;

Null_Address: constant Address := null;

type Address_Value is range -(2**31)..(2**31)-1;

function Location is new Unchecked_Conversion (Address_Value, Address);

function Label (Name: String) return Address;
pragma Interface (META, Label);
```

```
function ">" (Left, Right: Address) return Boolean;
pragma Interface (META, ">");

function "<" (Left, Right: Address) return Boolean;
pragma Interface (META, "<");

function ">=" (Left, Right: Address) return Boolean;
pragma Interface (META, ">=");

function "<=" (Left, Right: Address) return Boolean;
pragma Interface (META, "<=");

function "+" (Left: Address;       Right: Address_Value) return Address;
function "+" (Left: Address_Value; Right: Address)       return Address;
pragma Interface (META, "+");

function "-" (Left: Address;       Right: Address_Value) return Address;
function "-" (Left: Address;       Right: Address) return Address_Value;
pragma Interface (META, "-");


--================
- CALL SUPPORT
--================

type Subprogram_Value IS
record
   Proc_addr    : Address;
   Parent_frame : Address;
end record;

procedure Call (Subprogram: Subprogram_Value);
procedure Call (Subprogram: Address);

pragma Interface (META, Call);

Max_Object_Size  : CONSTANT := Max_Int;
Max_Record_Count : CONSTANT := Max_Int;
Max_Text_Io_Count : CONSTANT := Max_Int-1;
Max_Text_Io_Field : CONSTANT := 1000;


private
  type Memory is
  record
   null:
  end record;

end System;
```

Appendix F, Cont.

4. Restrictions on Representation Clauses

The hardware needs a minimum of 32 bits to represent floating point and access types. Therefore, specifying a size of less than 32 bits cannot be handled simply by the underlying hardware (LRM 13.1 (10)).

The Compiler supports the following representation clauses:

Length Clauses: for enumeration and derived integer types 'SIZE
        attribute (LRM 13.2(a))

Length Clauses: for composite types 'SIZE attribute (LRM 13.2(a)
        Size clauses for composite types are rejected when the
        specified size implies compression of composite components.

Length Clauses: for access types 'STORAGE_SIZE attribute (LRM 13.2(b))

Length Clauses: for tasks types 'STORAGE_SIZE attribute (LRM 13.2(c))

Length Clauses: for fixed point types 'SMALL attribute (LRM 13.2(d))

Enumeration Clauses: for character and enumeration types other than
        boolean (LRM 13.3)

Record representation Clauses (LRM 13.4) with following constraints:
        - Each component of the record must be specified with
        a component clause.
        - The alignment of the record is restricted to mod 2.
        word (16 bit)aligned.
        - Bits are ordered right to left within a byte.

Address Clauses: for objects. entries and external subprograms
        (pragma INTERFACE used) (LRM 13.5(a)(c))

This compiler does NOT support the following representation clauses:

Enumeration Clauses: for boolean (LRM 13.3)

Address Clauses: for packages, task units, and non-external Ada
        subprograms (LRM 13.5(b))

Appendix F, Cont.

5. Implementation dependent naming conventions

There are no implementation-generated names denoting implementation dependent components.

6. Interpretation of Expressions in Address Clause

Expressions that appear in address specifications are interpreted as the first storage unit of the object.

7. Restrictions on Unchecked Conversions

Unchecked conversions are allowed between any types or subtypes unless the target type is an unconstrained record or array type.

8. I/O Package Characteristics

Sequential_IO can be instantiated for unconstrained array types or unconstrained types with discriminants without default values, but not Direct_IO.

In TEXT_IO the type COUNT is defined as follows:

    type COUNT is range 0 .. 2_147_483_646;

In TEXT_IO the subtype FIELD is defined as follows:

    subtype FIELD is INTEGER range 0..1000;

In TEXT_IO, the Form parameter of procedures Create and Open is not supported. (If you supply a Form parameter with either procedure. it is ignored.)

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	199 * 'A' & '1'
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	199 * 'A' & '2'
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except	100 * 'A' & '3' & 99 * 'A'

Name and Meaning   Value

    for a character near the middle.

$BIG_ID4   100 * 'A' & '4' & 99 * 'A'
    An identifier the size of the
    maximum input line length which
    is identical to $BIG_ID3 except
    for a character near the middle.

$BIG_INT_LIT   197 * '0' & "298"
    An integer literal of value 298
    with enough leading zeroes so
    that it is the size of the
    maximum line length.

$BIG_REAL_LIT   195 * '0' & "690.0"
    A universal real literal of
    value 690.0 with enough leading
    zeroes to be the size of the
    maximum line length.

$BIG_STRING1   '"' & 100 * 'A' & '"'
    A string literal which when
    catenated with BIG_STRING2
    yields the image of BIG_ID1.

$BIG_STRING2   '"' & 99 * 'A' & '1' & '"'
    A string literal which when
    catenated to the end of
    BIG_STRING1 yields the image of
    BIG_ID1.

$BLANKS   180 * ' '
    A sequence of blanks twenty
    characters less than the size
    of the maximum line length.

$COUNT_LAST   2_147_483_646
    A universal integer
    literal whose value is
    TEXT_IO.COUNT'LAST.

$DEFAULT_MEM_SIZE   2147483647
    An integer literal whose value
    is SYSTEM.MEMORY_SIZE.

$DEFAULT_STOR_UNIT   8
    An integer literal whose value
    is SYSTEM.STORAGE_UNIT.

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	TELEGEN2
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	1000
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	131_073.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	63
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BADCHAR*`/%
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/NONAME/DIRECTORY

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-131_073.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	200
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:" & 195 * '0' & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 193 * '0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 198 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	SHORT_SHORT_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	TELEGEN2
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

Name and Meaning   Value

$NEW_MEM_SIZE   2147483647
    An integer literal whose value
    is a permitted argument for
    pragma MEMORY_SIZE, other than
    $DEFAULT_MEM_SIZE. If there is
    no other value, then use
    $DEFAULT_MEM_SIZE.

$NEW_STOR_UNIT   8
    An integer literal whose value
    is a permitted argument for
    pragma STORAGE_UNIT, other than
    $DEFAULT_STOR_UNIT. If there is
    no other permitted value, then
    use value of SYSTEM.STORAGE_UNIT.

$NEW_SYS_NAME   TELEGEN2
    A value of the type SYSTEM.NAME,
    other than $DEFAULT_SYS_NAME. If
    there is only one value of that
    type, then use that value.

$TASK_SIZE   32
    An integer literal whose value
    is the number of bits required
    to hold a task object which has
    a single entry with one 'IN OUT'
    parameter.

$TICK   0.01
    A real literal whose value is
    SYSTEM.TICK.

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C   This test expects that the string "-- TOP OF PAGE.   -- 63" of line 204 will   appear at the top of the listing page   due to a pragma PAGE in line 203; but   line 203 contains text   that follows the pragma, and it is this that must   appear at the top of the page.

b. A39005G   This test unreasonably expects a component   clause   to pack an array component   into a minimum size (line 30).

c. B97102E   This test contains an unitended illegality: a   select statement contains a   null statement at the place of a selective wait alternative (line 31).

d. C97116A   This test contains race conditions, and it assumes that guards are evaluated   indivisibly.   A conforming   implementation may use interleaved execution in   such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING-_OF_THE_GUARD results · in a call to REPORT.FAILED at one   of lines 52 or 56.

e. BC3009B   This test wrongly expects that circular   instantiations will be detected in several compilation units even though none   of the   units is illegal with respect to the units it depends on;   by AI-00256,   the illegality need not be detected until execution   is attempted (line 95).

f. CD2A62D   This test wrongly requires that an array object's   size be no greater than 10   although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests]   These
    tests wrongly attempt to check the size of objects of a derived
    type   (for which a 'SIZE length clause is given) by passing them
    to a derived subprogram (which implicitly converts them to the
    parent type (Ada standard   3.4:14)). Additionally, they use the
    'SIZE length clause and attribute,   whose interpretation is
    considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests]   These tests
    assume that dependent tasks will terminate while the main pro-
    gram executes a loop that simply tests for task termination; this
    is not   the case, and the main program may loop indefinitely
    (lines 74, 85, 86 & 96,   86 & 96, and 58, resp.).

i. CD2B15C & CD7205C   These tests expect that a 'STORAGE_SIZE
    length clause provides precise control over the number of
    designated objects in a collection; the Ada standard   13.2:15
    allows that such control must not be expected.

j. CD2D11B   This test gives a SMALL representation clause for a
    derived fixed-point type   (at line 30) that defines a set of
    model numbers that are not necessarily   represented in the
    parent type; by Commentary AI-00099, all model numbers of   a
    derived fixed-point type must be representable values of the
    parent type.

k. CD5007B   This test wrongly expects an implicitly declared sub-
    program to be at the   the address that is specified for an un-
    related subprogram (line 303).

l. ED7004B, ED7005C & D, ED7006C & D [5 tests]   These tests check
    various aspects of the use of the three SYSTEM pragmas;   the AVO
    withdraws these tests as being inappropriate for validation.

m. CD7105A   This test requires that successive calls to CALENDAR.-
    CLOCK change by at   least SYSTEM.TICK; however, by Commentary
    AI-00201, it is only the expected   frequency of change that must
    be at least SYSTEM.TICK--particular instances   of change may be
    less (line 29).

n. CD7203B, & CD7204B   These tests use the 'SIZE length clause and
    attribute, whose interpretation   is considered problematic by
    the WG9 ARG.

o. CD7205D   This test checks an invalid test objective: it treats
    the specification of   storage to be reserved for a task's
    activation as though it were like the   specification of storage
    for a collection.

p. CE2107I   This test requires that objects of two similar scalar
    types be distinguished   when read from a file--DATA_ERROR is

expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

q. CE3111C   This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A   This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

s. CE3411B   This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

References and page numbers in this appendix are consistent with compiler documentation and not with this report.

In addition to the switches described in the sequel the following switches are available:

-W +enable_traceback
    allow internal compiler errors to propagate all the way out. The default is to inhibit internal exceptions call chains from being displayed.

-W -debug_info
    inhibits insertion of Ada symbolic debugger information into the working sublibrary. This will speed up compilation. The default is to insert all information into the working sublibrary.

-W +link_target=bbn
    generates a Berkeley Unix style link script. The default is to generate a System V style link script.

2.1. The Ada Compiler ("ada")

The TeleGen2 Ada Compiler is invoked by the *ada* command. Unless you specify otherwise, the front end, middle pass, and code generator are executed each time the compiler is invoked.

Before you can compile, you must (1) make sure you have access to TeleGen2, (2) create a library file, and (3) create a sublibrary. These steps were explained in the Getting Started section of the Overview. We suggest you review that section, and then compile, link, and execute the sample program as indicated before you attempt to compile other programs.

This section focuses specifically on command-level information relating to compilation, that is, on invoking the compiler and using the various options to control the compilation process. Details on the TeleGen2 compilation process and guidelines for using the compiler most effectively are in the Compiler chapter of the *User Guide* volume. (You might want to look at Figure 3-1 in that volume right now, to give you insight into the TeleGen2 compilation process and to see how the options mentioned in this Command Summary volume relate to the actual compilation process.)

The syntax of the command to invoke the Ada compiler is:

```
ada {<"common_option">}  {<option>}  <input_spec>
```

where:

<"common_option">	None or more of the following set of options that are common to many TeleGen2 commands:
	
	    −l(ibfile or −t(emplib
	    −V(space_size
	    −v(erbose
	
	These options were discussed in Chapter 1.

<option>	None or more of the compiler-specific options discussed below.

<input_spec>	The Ada source file(s) to be compiled. It may be:
	
	  * One or more Ada source files, for example:
	     /user/john/example
	     Prog_A.text
	     ciosrc/calc_mem.ada
	     calcio.ada myprog.ada
	    *.ada
	  ■ A file containing names of files to be compiled. Such a file must have the extension ".ilf". You can find details for using input-list files in the *User Guide* portion of your TeleGen2 documentation set.
	  ■ A combination of the above.

Please note that the compiler defaults are set for your convenience. In most cases you will not need to use additional options: a simple "ada <input_spec>" is sufficient. However, options

are included to provide added flexibility. You can, for example, have the compiler quickly check the source for syntax and semantic errors but not produce object code (−e(rrors_only) or you can compile, bind, and link an main program with a single compiler invocation (−m(ain). Other options are provided for other purposes.

The options available with the *ada* command, and the relationships among them, are illustrated in the following figure. Remember that each of the options listed is identified by a dash followed by a single letter (e.g., "−e"). The parenthesis and the characters following the option are for descriptive purposes only; they are not part of the option.

```
                          ┌─────┐
                          │ ada │
                          └─────┘
          ┌──────────────────┴──────────────────────┐
  −l(ibfile <libname>      │      −t(emplib <sublib>{,...}
          └──────────────────┬──────────────────────┘
                   −V(space_size 2000
                          │
                      −v(erbose
                          │
          ┌──────────────────┴──────────────────────┐
  −e(rrors_only          │     ( compile to object )
                          │          −d(ebug
                          │      −i(nhibit <key>†
                          │          −k(eep
                          │      −O(ptimize <key>†
                          │        −S(ource_asm
                          │     −u(pdate_lib <key>†
                          │      −x(ecution_profile
          └──────────────────┬──────────────────────┘
                    −C(ontext 1
                          │
                  −E(rror_abort 999
          ┌──────────────────┴──────────────────────┐
  −L(ist                 │      −F(ile_only_errs
          └──────────────────┬──────────────────────┘
                □ −s(oftware_float
                          │
                   −m(ain <unit>
                    ┌──────────────┐
                    │ <input_spec> │
                    └──────────────┘
```

---
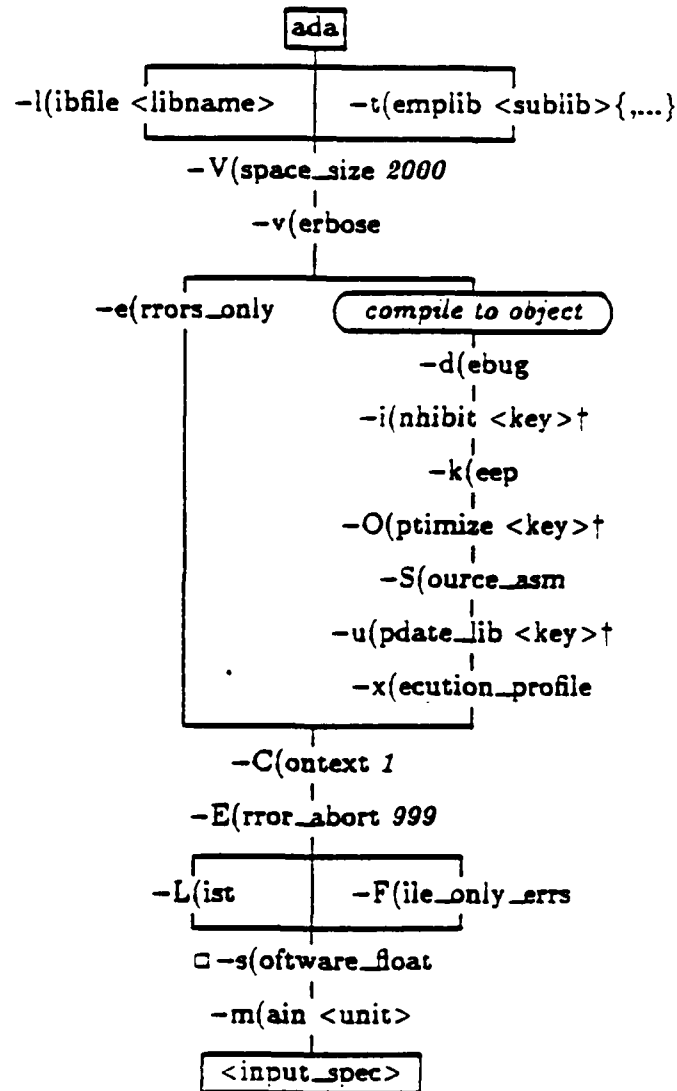
† (1) <key> for −O: refer to sect. (2) <key> for −u: i or s: s is the default. (3) <key> for −e a or certain combinations of lac.

The options available with the *ada* command are summarized in Table 2-2. The default situation (that is, what happens if the option is not used) is explained in the middle column. Each option is described in the paragraphs that follow the table.

Table 2-2. Summary of Compiler Options

Option	Default	Discussed in Section
Common options:		
−l(ibfile <libname>	Use liblst.alb as the library file.	1.4.1
−t(emplib <sublib...>	None	1.4.1
−V(space_size <value>	Set size to 2000 Kbytes.	1.4.2
−v(erbose	Do not output progress messages.	1.4.3
−d(ebug	Do not include debug information in object code. (−d sets −k(eep.)	2.1.1
−E(rror_abort <value>	Abort compilation after 999 errors.	2.1.2
−e(rrors_only	Run middle pass and code generator, not just front end.	2.1.3
−i(nhibit <key>†	Do not suppress run-time checks, source line references, or subprogram name information in object.	2.1.4
−k(eep	Discard intermediate representations of secondary units.	2.1.5
−m(ain <unit>	Do not produce executable code (binder/linker not executed).	2.1.6
−O(ptimize <key>†	Do not optimize code.	2.1.7
−s(oftware_float	Use hardware floating-point support.	2.1.8
−u(pdate_lib <key>†	Do not update library when errors are found (multi-unit compilations).	2.1.9
−x(ecution_profile	Do not generate execution-profile code.	2.1.10
Listing options:		
−C(ontext <value>	Include 1 line of context with error message.	2.1.11.1
−L(ist	Do not generate a source-error listing.	2.1.11.2
−F(ile_only_errs	Do not generate an errors-only listing. only.	2.1.11.3
−S(ource_asm	Do not generate assembly listing.	2.1.11.4

2.1.1. −d(ebug − Generate Debugger Information. The code generator must generate special information for any unit that is to be used with the TeleGen2 symbolic debugger. The generation of this information is enabled by use of the −d option. The use of −d automatically

† (1) <key> for −O: refer to sept. (2) <key> for −u: i or s is the default. (2) <key> for −c a or certain combinations of lac.

sets the −k(eep option. This to make sure that the High Form, Low Form, and debugger information for secondary units are not deleted.

To see if a unit has been compiled with the −d(ebug option, use the *als* command with the −X(tended option. Debugger information exists for the unit if the "dbg_info" attribute appears in the listing for that unit. The default situation is that no debugger information is produced.

Performance note. While the compilation time overhead generated by the use of −d(ebug is minimal, retaining this optional information in the Ada library increases the space overhead.

2.1.2. −E(rror_abort − Set an Error Count for Aborting Compilation.

The compiler maintains separate counts of all syntactic errors, semantic errors, and warning messages detected by the front end during a compilation.

A large number of errors generally indicates that errors in statements appearing earlier in the unit have "cascaded" through the rest of the code. A classic example is an error occurring in a statement that declares a type. This causes subsequent declarations that use the type to be in error, which further causes all statements using the declared objects to be in error. In such a situation, only the first error message is useful. Aborting the compilation at an early stage is therefore often to your advantage; the −E option allows you to do it.

The format of the option is:

   −E   <value>

where <value> is the number of errors or warnings allowed. The default value is 999. The minimum value is 1. Caution: If you do not use the −E option, it is possible to have 999 warning messages *plus* 999 syntax errors *plus* 999 semantic errors without aborting compilation, since each type of error is counted separately.

2.1.3. −e(rrors_only − Check Source But Don't Generate Code.

This option instructs the compiler to perform syntactic and semantic analysis of the source program without generating Low Form and object code. That is, it calls the front end only, not the middle pass and code generator. (This means, of course, that only front end errors are detected and that only the High Form intermediates are generated.) This option is typically used during early code development where execution is not required and speed of compilation is important.

Note: Although High Form intermediates are generated with the −e option, these intermediates are deleted at the end of compilation. This means that the library is not updated.

The −e option cannot be used with −S(ource_asm, since the latter requires the generation of object code. If −e is not used (the default situation), the source is compiled to object code (if no errors are found). The −e option is also incompatible with −k(eep, −d(ebug, −O(ptimize, and other options that require processing beyond the front end phase of compilation.

2.1.4. −i(nhibit − Suppress Checks and Source Information.

The −i(nhibit option allows you to suppress, within the generated object code, certain run-time checks, source line references, and subprogram name information.

The Ada language requires a wide variety of *run-time checks* to ensure the validity of operations. For example, arithmetic overflow checks are required on all numeric operations, and range checks are required on all assignment statements that could result in an illegal value being assigned to a variable. While these checks are vital during development and are an important asset of the language, they introduce a substantial overhead. This overhead may be prohibitive

in time–critical applications.

Although the Ada language provides pragma Suppress to selectively suppress classes of checks. using the pragma requires you to modify the Ada source. The −i(nhibit option provides an alternative mechanism.

The compiler by default stores *source line and subprogram name information* in the object code. This information is used to display a source level traceback when an unhandled exception propagates to the outer level of a program: it is particularly valuable during development, since it provides a direct indication of the source line at which the exception occurs and the subprogram calling chain that led to the line generating the exception.

The inclusion of source line information in the object code, however. introduces an overhead of 6 bytes for each line of source that causes code to be generated. Thus, a 1000-line package may have up to 6000 bytes of source line information. For one compilation unit, the extra overhead (in bytes) for subprogram name information is the total length of all subprogram names in the unit (including middle pass-generated subprograms), plus the length of the compilation unit name. For space-critical applications, this extra space may be unacceptable; but it can be suppressed with the −i(nhibit option. When source line information is suppressed, the traceback indicates the offset of the object code at which the exception occurs instead of the source line number. When subprogram name information is suppressed. the traceback indicates the offsets of the subprogram calls in the calling chain instead of the subprogram names. (For more information on the traceback function, refer to the Programming Guide chapter in your *Reference Information* volume.)

The format of the −i(nhibit option is:

    −i <suboption>{<suboption>}

where <suboption> is one or more of the single-letter suboptions listed below. Combinations of suboptions are possible. When more than one suboption is used, the suboptions appear together with no separators. For example, "−i lnc".

l(line_info)	Suppress source line information in object code.
n(ame_info)	Suppress subprogram name information in object code.
c(hecks)	Suppress run-time checks — elaboration. overflow, storage access, discriminant. division. index, length, and range checks.
a(ll)	Suppress source line information. subprogram name information. and run-time checks. In other words. a (=inhibit all) is equivalent to lnc.

As an example of use, the command....

    ada −v −i lc my_file.ada

...inhibits the generation of source line information and run-time checks in the object code of the units *my_file.ada*.

2.1.5. −k(eep − Retain Intermediate Forms. As a default, the compiler deletes the High Form and Low Form intermediate representations of all compiled secondary units from the working sublibrary. Deletion of these intermediate forms can significantly decrease the size of sublibraries − typically 50% to 80% for multi-unit programs. On the other hand, some of the information within the intermediate forms may be required later. For example, High Form is required if the unit is to be referenced by the Ada Cross-Referencer (*axr*). In addition, information required by the debugger and the Global Optimizer must be saved if these utilities are used. For these reasons. the −*k* option is provided with the *ada* command. The −*k* option:

- Must be used if the compiled unit is to be optimized later with *aopt*; otherwise, *aopt* issues an error message and the optimizer aborts.

- Should be used if the unit is to be cross-referenced later; otherwise. an error message is issued when the Ada Cross-Referencer attempts to cross-reference that unit.

- Need not be used with −d(ebug, since −*k* is set automatically whenever −*d* is used.

To verify that a unit has been compiled with the −k(eep option (has not been "squeezed"), use the *als* command with the −X(tended option. A listing will be generated that shows whether the intermediate forms for the unit exist. A unit has been compiled with −k(eep if the attributes high_form and low_form appear in the listing for that unit.

2.1.6. −m(ain − Compile a Main Program. This option tells the compiler that the unit specified with the option is to be used as a main program. After all files named in the input specification have been compiled, the compiler invokes the prelinker (binder) and the native linker by calling *ald* to bind and link the program with its extended family. An executable file named <unit> is left in the current directory. The linker may also be invoked directly by the user with the *ald* command.

The format of the option is:

    −m <unit>

where <unit> is the name of the main unit for the program. If the main unit has already been compiled, it does not have to be in the input file. However, the body of the main unit, if previously compiled, must be present in the current working sublibrary.

Note: Options specific to the linker (invoked via *ald*) may be specified on the *ada* command line when the −*m* option is used. With −*m*. the compiler will call *ald* when compilation is complete, passing to it *ald*-specific options specified with the *ada* command. For example...

    ada −m welcome   −T 2   −o new   sample.ada

...instructs the compiler to compile the Ada source file. *sample.ada*. which contains the main program unit Welcome. After the file has been compiled. the compiler calls the linker, passing to it the −*T* and −*o* options with their respective arguments. The linker produces an executable version of the unit, placing it in file *new* as requested by the −*o* option.

2.1.7. −O(ptimize − Optimize Object Code. This option causes the compiler to invoke the global optimizer to optimize the Low Form generated by the middle pass for the unit being compiled. The code generator takes the optimized Low Form as input and produces more efficient object code. The format of this option is:

    −O <key>

where <key> is at least one of the optimizer suboption keys discussed in the Global Optimizer chapter. Please refer to that chapter for all information regarding the use of the optimizer. The chapter discusses using the optimizer as a standalone tool for collections of compiled but unoptimized units <u>and</u> using the −O(ptimize option with the *ada* command. The latter topic includes a definition of the −O(ptimize suboption key values plus a presentation of two other *ada* options (−G(raph and −I(nline_list, not shown on the *ada* chart) that may be used in conjunction with the −O(ptimize option. *Note:* We strongly recommend that you do not attempt to use the optimizer until the code being compiled has been fully debugged and tested.

2.1.8. −s(oftware_float − Use Software Floating-Point Support. *This option may not be available with your TeleGen2 system; please consult the Overview portion to see if it is provided.* The Ada linker selects hardware floating-point support by default. If you do not have hardware floating point support or if you wish to generate code compatible with such machines, use the −*s* option. <u>In addition:</u> If you use the −*s* option, the library file you use for compilation must contain the the name of the *software* floating point run-time sublibrary, *s_rt.sub*. Refer to the Library Manager chapter in your *User Guide* volume for more information on the run-time sublibrary.

2.1.9. −u(pdate_lib − Update the Working Sublibrary. The −u(pdate_lib option tells the compiler when to update the library. It is most useful for compiling multiple source files. The format of the option is:

      −u <key>

where <key> is either "s" (source) or "i" (invocation).

  i   "i" tells the compiler to update the working sublibrary after all files submitted in that invocation of *ada* have compiled successfully. If an error is encountered, the library is not updated. even for source files that compile successfully. In addition, all remaining source files will be compiled for syntactic and semantic errors only. *Implications:* (1) If an error exists in any source file you submit, the library will not be updated, even if all other files are error free. (2) Compilation is faster, since the library is updated only once. at the end of compilation.

  s   (This is the default; it is equivalent to not using the −u(pdate_lib option at all.) "s" tells the compiler to update the library after all units within a single source file compile successfully. If the compiler encounters an error in any unit within a source file, all changes to the working sublibrary for the erroneous unit and for all other units in the file are discarded. However. library updates for units in previous or remaining source files are unaffected. *Implications:* (1) You can submit files containing possible errors and still have units in other files compile successfully into the library. (2) Compilation is slightly slower, since the library is updated once for each file.

Therefore:

> Use "u s" (or no −u(pdate option) when:
>
> > You're not sure all units will compile successfully.
> > Compilation speed is not especially important.
>
> Use "u i" when:
>
> > You are reasonably certain your files will compile successfully.
> > Fast compilation is important.

2.1.10. −x(ecution_profile — Generate Profile Information. The −x(ecution_profile option uses the code generation phase of compilation to place special information in the generated code that can be used to obtain a "profile" of a program's execution. This information is generated by a facility known as "the profiler." Refer to your *User Guide* volume for information on how to use the profiler to obtain execution timing and subprogram call information for a program.

Important: If any code in a program has been compiled with the −x(ecution_profile option, that option must also be used with *ald* when the program is bound and linked. Otherwise, linking aborts with an error such as "Undefined RECORD$$CURRENT".

2.1.11. Listing Options. The listing options specify the content and format of listings generated by the compiler. Assembly code listings of the generated code can also be generated.

2.1.11.1. −C(ontext — Include Source Lines Around the Error. When an error message is sent to *stderr*, it is helpful to include the lines of the source program that surround the line containing the error. These lines provide a context for the error in the source program and help to clarify the nature of the error. The −C option controls the number of source lines that surround the the error.

> The format of the option is:

> −C <value>

where <value> is the number of source context lines output for each error. The default for <value> is 1. This parameter specifies the total number of lines output for each error (including the source line that contains the error). The first context line is the one immediately before the line in error; other context lines are distributed before and after the line in error. Let's say that *trialprog.ada*, which consists of the following text...

```
package P is
  type T1 is range 1..10;
  type T2 is digits 1;
  type Arr is array (1..2) of integer; type T3 is new Arr;    -- OK.

    package Inner is
      type In1 is new T1;      -- ERROR: T1 DERIVED.
      type In2 is new T2;      -- ERROR: T2 DERIVED.
      type In3 is new T3;      -- ERROR: T3 DERIVED.
      type Inarr is new Arr;   -- OK.

    end Inner;

end P;
```

...were compiled as follows:

    ada   -e -C 2   trialprog.ada

(The —e option here is used for error checking and —C(ontext is set to 2 to display two lines of source.) The output produced would look like this:

```
       7:        package Inner is
       8:           type In1 is new T1;      -- ERROR: T1 DERIVED.
                    .......
    >>>   Illegal parent type for derivation <3.4:15,7.4.1:4>


       8:           type In1 is new T1;      -- ERROR: T1 DERIVED.
       9:           type In2 is new T2;      -- ERROR: T2 DERIVED.
                    .......
    >>>   Illegal parent type for derivation <3.4:15,7.4.1:4>


       9:           type In2 is new T2;      -- ERROR: T2 DERIVED.
      10:           type In3 is new T3;      -- ERROR: T3 DERIVED.
                    .......
    >>>   Illegal parent type for derivation <3.4:15,7.4.1:4>
```

2.1.11.2. —L(ist — Generate a Source Listing. This option instructs the compiler to output a listing of the source being compiled. interspersed with error information (if any). The listing is output to <file_spec>.l, where <file_spec> is the name of the source file (minus the extension). If <file_spec>.l already exists. it is overwritten.

If input to the *ada* command is an input-list file (<file_spec>.ilf), a separate listing file is generated for each source file listed in the input file. Each resulting listing file has the same name as the parent file, except that the extension ".l" is appended. Errors are interspersed with the listing. If you do not use —L (the default situation), errors are sent to *stdout* only; no listing is produced. - L is incompatible with —F.

2.1.11.3. −F(ile_only_errs - Put Only Errors in Listing File. This option is used to produce a listing containing only the errors generated during compilation; source is not included. The output is sent to <file_spec>.l. −F is incompatible with −L.

2.1.11.4. −S(ource_asm − Generate a Source/Assembly Listing. This option instructs the compiler to generate an assembly listing and send it to a file named <unit>.<ext>, where <unit> is the name of the unit in the user-supplied source file and <ext> is the file extension (it may be "s" or something else, depending on your configuration). The listing consists of assembly code intermixed with source code as comments. If input to the *ada* command is an input-list file (<file_spec>.ilf), a separate assembly listing file is generated for each unit contained in each source file listed in the input file. If −S is not used (the default situation), an assembly listing is not generated.

2.2. The Ada Linker ("ald")

The TeleGen2 Ada Compiler produces object code from Ada source code. The TeleGen2 Ada Linker takes the object (of a main program) that is produced by the compiler and produces a UNIX executable module. The TeleGen2 Ada Linker will be called "the linker" in the remainder of this manual.

To produce executable code, the linker (1) generates elaboration code and a link script (this is called "binding" or "prelinking") then (2) calls the UNIX link editor (*ld*) to complete the linking process.

The linker is invoked with the *ald* command; it can also be invoked with the −m(ain option of the *ada* command. In the latter case the compiler passes appropriate options to the linker, to direct its operation.

In the simplest case, the *ald* command takes one argument − the name of the main unit of the Ada program structure that is to be linked − and produces one output file − the executable file produced by the linking process. The executable file is placed in the directory where *ald* was executed, under the name of the main unit used as the argument to *ald*. For example, the command

        ald   main

links the object modules of all the units in the extended family of the unit Main. The name of the resulting executable file will simply be "main". Important: When using the *ald* command, the body of the main unit to be prelinked must be in the working sublibrary.

The general syntax of the *ald* command is:

```
ald  {<"common_option">}  {<option>}  <unit>
```

where:

<"common_option">	None or more of the following set of options that are common to many TeleGen2 commands: −l(ibfile or −t(emplib −V(space_size −v(erbose These options were discussed in Chapter 1.
<option>	None or more of the options discussed in the following sections.
<unit>	The name of the main unit of the Ada program to be linked.

The options available with the *ald* command and the relationships among them are illustrated below.

```
                            ┌─────┐
                            │ ald │
                            └─────┘
        ┌──────────────────────┼──────────────────────────┐
  −l(ibfile <libname>          │        −t(emplib <sublib>{,...}
        └──────────────────────┼──────────────────────────┘
                      −V(space_size 2000
                            │
                         −v(erbose
                            │
                        −b(ind_only
                            │
                     −o(utput <file_spec>
                            │
                    −P(ass_options 'string'
                            │
                    −p(ass_objects 'string'
                            │
                    □ −s(oftware_float
                            │
                     −S("asm_listing"
                            │
                     −T(raceback 15
                            │
                    −x(ecution_profile
        ┌──────────────────────┼──────────────────────────┐
  −D(elay_NonPreempt           │        −w("timeslice" 0
        └──────────────────────┼──────────────────────────┘
                     −X(ception_show
                            │
                   −Y 8192 |bytes-long|
                            │
                   −y 1024 |bytes-natural|
                       ┌─────────┐
                       │ <unit>  │
                       └─────────┘
```
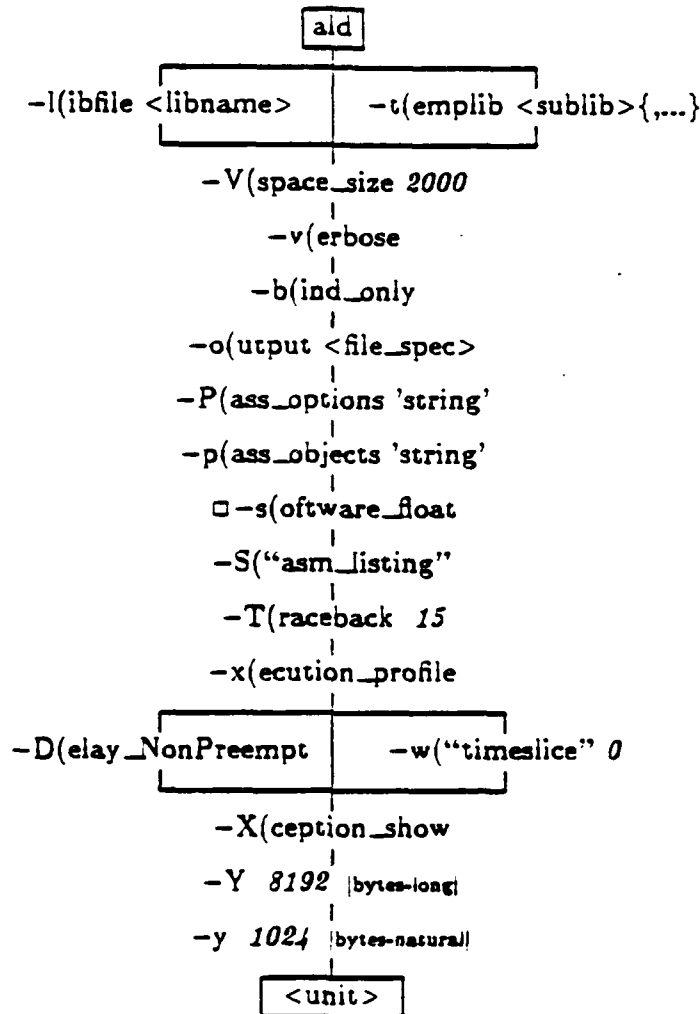
2.2.1. −b(ind_only − Produce Elaboration Code Only. To provide you with more control over the linking process, the *−b* option causes the linker to abort after it has created the elaboration code and the linking order, but before invoking the UNIX link editor. This option allows you to edit the link order for special applications and then invoke the link editor directly. The link order is contained in an executable script that invokes the link editor with the appropriate options and arguments. The name of the script produced is <unit>.lnk, which is placed in your working directory. To complete the link process, enter "<unit>.lnk".

The name of the file containing the elaboration code is <unit>.obm, which is placed in the object directory of the working sublibrary.

For System V versions of UNIX, the file names generated as a result of linking are created by appending the 2-letter extension to the unit name and truncating the result to 14 characters.

2.2.2. −o(utput − Name the Output File. This option allows you to specify the name of the output file produced by the linker. For example, the command...

        ald −o yorkshire main

...causes the linker to put the executable module in the file *yorkshire*.

2.2.3. -P(ass_Options - **Pass Options to the Linker.** This option allows you to pass a string of options directly to the UNIX link editor. For example, the command

        ald   -P '-t -r" main

adds the string "-t -r" to the options of the link editor when it is invoked. The options must be quoted (double or single quotes).

2.2.4. -p(ass_objects - **Pass Arguments to the Linker.** This option allows you to pass a string of arguments directly to the UNIX link editor. For example, the command

        ald   -p 'cosine.o /usr/lib/libm.a' main

causes the link editor to link the object file *cosine.o* (which it expects to find in the current working directory), and to search the library */usr/lib/libm.a* for unresolved symbol references. (The location of the *libm.a* library may be different on your system.) Remember that the link editor searches a library exactly once at the point it is encountered in the argument list, so references to routines in libraries must occur before the library is searched. That is, files that include references to library routines must appear before the corresponding libraries in the argument list. Objects and archives added with the *-p* option will appear in the linking order after Ada object modules and run-time support libraries, but before the standard C library (*/lib/libc.a*). This library is always the last element of the linking order.

    You can also use the *-p* option to specify the link editor's *-l* option, which causes the link editor to search libraries whose names have the form "/lib/libname.a" or "/usr/lib/libname.a". For example, the command

        ald   -p '-lxyz'

causes the link editor to search the directories */lib* and */usr/lib* (in that order) for file *libxyz.a*.

2.2.5. -S("asm_listing" - **Produce an Assembly Listing.** The *-S* option is used to output an assembly listing from the elaboration process. The output is put in a file, <file>.obm.s, where <file> is the name of the main unit being linked. (The file extension may be different on your system.)

2.2.6. -s(oftware_float - **Use Software Floating-Point Support.** *This option may not be available on your TeleGen2 system. Please consult the Overview portion of this volume to see if it is provided.* The Ada linker currently selects hardware floating-point support by default. This default situation is provided for users of systems with an arithmetic coprocessor. If you do not have hardware floating point support or if you wish to generate code compatible with such machines, use the *-s* option. *In addition:* if you use the *-s* option, the library file you use for compiling and linking must contain the name of the *software* floating point run-time sublibrary, *s_rt.sub*. Refer to the Library Manager chapter in your *User Guide* volume for more information on the run-time sublibrary.

2.2.7. -T(raceback - **Set Levels for Tracing Exceptions.** When a run-time exception occurs (and is not handled by an exception handler), the name and line number of the unit where the exception occurred is displayed along with a recursive history of the units which called that unit. (See the "Exception Handling" section in the Programming Guide chapter of your *Reference Information* volume for a more complete explanation of exception reports.) The *-T* option allows you to set the number of levels in this recursive history. For example, the

command

      ald   −T 3 main

specifies that traceback histories will be three levels deep. The default value for this option is 15.

When an exception occurs, the run-time support system stores the history in a preallocated block of memory. Since the size of this block is determined by the −T option, setting this value to a large number can introduce objectionable overhead in deeply nested, time-critical code. You may wish to make this value smaller for well-tested programs.

2.2.8. **−x(ecution_profile − Bind and Link for Profiling.** This option is used for units that have been compiled with the −x option. Use of −x with *ada* causes the code generator to include, in the object, special code that will later be used to provide a profile of the program's execution.

If −x is used with *ada*, it must be used with *ald* as well. The −x option of *ald* instructs the linker to link in the profiling run-time support routines and generate a subprogram dictionary, *profile.dic*, for the program. The dictionary is a text file containing the names and addresses of all subprograms in the program. The dictionary can be used to produce a listing showing how the program executes.

Refer to the Ada Profiler chapter in your *User Guide* volume for a full discussion of the profiler.

2.2.9. Tasking Options. The following *ald* options are binding options used for task execution. They are therefore useful only for linking programs that contain tasking code.

2.2.9.1. −D(elay_NonPreempt − Specify Non-Preemptive Delay. By default, the TeleGen2 run-time is set for preemptive delay handling. That is, an active task is preempted if another task is waiting that has a priority equal to or greater than that of the active task.

The −D option allows you to specify *non*-preemptive delay handling. With non-preemptive delay, a task is scheduled only when a synchronization point is reached. −D(elay_NonPreempt is incompatible with the −w option (see below).

2.2.9.2. −w("timeslice" − Limit Task Execution Time. The −w option allows you to define the maximum time a task may execute before it is rescheduled. The format of the option is:

      −w <value>

where <value> is the maximum time the task is to execute, in milliseconds, before a task switch occurs between it and a task having the same or higher priority. The default value is 0 (no timeslice). If you choose a value greater than 0, it must be at least as great as the clock interval time.

Since rescheduling of tasks is incompatible with interrupt-scheduling, −w is incompatible with −D(elay_NonPreempt (see above).

2.2.9.3. −X(ception_show − Report Unhandled Exceptions. By default, unhandled exceptions that occur in tasks are not reported: instead, the task terminates silently. The −X option allows you to specify that such exceptions are to be reported. The output is similar to that displayed when an unhandled exception occurs in a main program.

2.2.9.4. −Y and −y - Alter Stack Size. In the absence of a representation specification for task storage_size, the run time will allocate 8192 bytes of storage for each executing task. You can change the amount of space allocated for tasking by using the − Y and −y options.

−Y specifies the size of the basic task stack. The format of the option is:

−Y <value>

where <value> is the size of the task stack in 32-bit (long_integer) bytes. The default is 8192.

−y specifies the stack-guard size. The stack-guard space is the amount of additional space allocated per task to accommodate interrupts and exception-handling operations. The format of the option is:

−y <value>

where <value> is the size of the stack-guard size in 16-bit (natural) bytes. The value given must be greater than the task-stack size. The default is 1052 bytes; this is the amount allocated unless otherwise specified.

A representation specification for task storage size overrides a value supplied with either option.